Contexts, Local Variables, and Loops

Viktor Kunčak

Properties of Program Contexts

Some Properties of Relations

$$(p_1 \subseteq p_2) \rightarrow (p_1 \circ p) \subseteq (p_2 \circ p)$$

$$(p_1 \subseteq p_2) \rightarrow (p \circ p_1) \subseteq (p \circ p_2)$$

$$(p_1 \subseteq p_2) \wedge (q_1 \subseteq q_2) \quad \rightarrow \quad (p_1 \cup q_1) \subseteq (p_2 \cup q_2)$$

$$(p_1 \cup p_2) \circ q = (p_1 \circ q) \cup (p_2 \circ q)$$

Monotonicity of Expressions using $\cup$ and $\circ$

Consider relations that are subsets of $S \times S$ (i.e. S^2)
The set of all such relations is

$$C = \{r \mid r \subseteq S^2\}$$

Let $E(r)$ be given by any expression built from relation r and some additional relations $b_1, \ldots, b_n$, using $\cup$ and $\circ$.
Example: $E(r) = (b_1 \circ r) \cup (r \circ b_2)$
$E(r)$ is function $C \to C$, maps relations to relations
Claim: E is monotonic function on C:

$$r_1 \subseteq r_2 \to E(r_1) \subseteq E(r_2)$$

Prove of disprove.

Monotonicity of Expressions using $\cup$ and $\circ$

Consider relations that are subsets of $S \times S$ (i.e. S^2)
The set of all such relations is

$$C = \{r \mid r \subseteq S^2\}$$

Let $E(r)$ be given by any expression built from relation r and some additional relations $b_1, \ldots, b_n$, using $\cup$ and $\circ$.
Example: $E(r) = (b_1 \circ r) \cup (r \circ b_2)$
$E(r)$ is function $C \to C$, maps relations to relations
Claim: E is monotonic function on C:

$$r_1 \subseteq r_2 \to E(r_1) \subseteq E(r_2)$$

Prove of disprove.
Proof: induction on the expression tree defining E, using monotonicity properties of $\cup$ and $\circ$

Union-Distributivity of Expressions using $\cup$ and $\circ$

Claim: E distributes over unions, that is, if $r_i, i \in I$ is a family of relations,

$$E(\bigcup_{i \in I} r_i) = \bigcup_{i \in I} E(r_i)$$

Prove or disprove.

Union-Distributivity of Expressions using $\cup$ and $\circ$

Claim: E distributes over unions, that is, if $r_i, i \in I$ is a family of relations,

$$E(\bigcup_{i \in I} r_i) = \bigcup_{i \in I} E(r_i)$$

Prove or disprove.
False. Take $E(r) = r \circ r$ and consider relations r_1, r_2. The claim becomes

$$(r_1 \cup r_2) \circ (r_1 \cup r_2) = r_1 \circ r_1 \cup r_2 \circ r_2$$

that is,

$$r_1 \circ r_1 \cup r_1 \circ r_2 \cup r_2 \circ r_1 \cup r_2 \circ r_2 = r_1 \circ r_1 \cup r_2 \circ r_2$$

Taking, for example, $r_1 = \{(1,2)\}$, $r_2 = \{(2,3)\}$ we obtain

$$\{(1,3)\} = \emptyset \quad (\textit{false})$$

Union "Distributivity" in One Direction

Lemma:

$$E(\bigcup_{i \in I} r_i) \supseteq \bigcup_{i \in I} E(r_i)$$

Union "Distributivity" in One Direction

Lemma:

$$E\left(\bigcup_{i \in I} r_i\right) \supseteq \bigcup_{i \in I} E(r_i)$$

Proof. Let $r = \bigcup_{i \in I} r_i$. Note that, for every i, $r_i \subseteq r$. We have shown that E is monotonic, so $E(r_i) \subseteq E(r)$. Since all $E(r_i)$ are included in $E(r)$, so is their union, so

$$\bigcup E(r_i) \subseteq E(r)$$

as desired.

Union-Distributivity - Refined

Does distributivity

$$E(\bigcup_{i \in I} r_i) = \bigcup_{i \in I} E(r_i)$$

hold, for each of these cases

1. If $E(r)$ is given by an expression containing r at most once?

Union-Distributivity - Refined

Does distributivity

$$E(\bigcup_{i \in I} r_i) = \bigcup_{i \in I} E(r_i)$$

hold, for each of these cases

1. If $E(r)$ is given by an expression containing r at most once? Proof: Induction on expression for $E(r)$. Only one branch of the tree may contain r. Note previous counter-example uses r twice.

Union-Distributivity - Refined

Does distributivity

$$E(\bigcup_{i \in I} r_i) = \bigcup_{i \in I} E(r_i)$$

hold, for each of these cases

1. If $E(r)$ is given by an expression containing r at most once? Proof: Induction on expression for $E(r)$. Only one branch of the tree may contain r. Note previous counter-example uses r twice.
2. If $E(r)$ contains r any number of times, but I is a set of natural numbers and r_i is an increasing sequence: $r_1 \subseteq r_2 \subseteq r_3 \subseteq \ldots$

Union-Distributivity - Refined

Does distributivity

$$E(\bigcup_{i \in I} r_i) = \bigcup_{i \in I} E(r_i)$$

hold, for each of these cases

1. If $E(r)$ is given by an expression containing r at most once? Proof: Induction on expression for $E(r)$. Only one branch of the tree may contain r. Note previous counter-example uses r twice.
2. If $E(r)$ contains r any number of times, but I is a set of natural numbers and r_i is an increasing sequence: $r_1 \subseteq r_2 \subseteq r_3 \subseteq \ldots$ Induction. In the previous counter-example the largest relation will contain all other $r_i \circ r_j$.
3. If $E(r)$ contains r any number of times, but $r_i, i \in I$ is a **directed family** of relations: for each i, j there exists k such that $r_i \cup r_j \subseteq r_k$, and I is possibly uncountably infinite.

Union-Distributivity - Refined

Does distributivity

$$E(\bigcup_{i \in I} r_i) = \bigcup_{i \in I} E(r_i)$$

hold, for each of these cases

1. If $E(r)$ is given by an expression containing r at most once? Proof: Induction on expression for $E(r)$. Only one branch of the tree may contain r. Note previous counter-example uses r twice.
2. If $E(r)$ contains r any number of times, but I is a set of natural numbers and r_i is an increasing sequence: $r_1 \subseteq r_2 \subseteq r_3 \subseteq \ldots$ Induction. In the previous counter-example the largest relation will contain all other $r_i \circ r_j$.
3. If $E(r)$ contains r any number of times, but $r_i, i \in I$ is a **directed family** of relations: for each i, j there exists k such that $r_i \cup r_j \subseteq r_k$, and I is possibly uncountably infinite. Induction. Generalizes the previous case.

Union-Distributivity Case of Increasing Sequence

$$E(\bigcup_{i\in I} r_i) = \bigcup_{i\in I} E(r_i)$$

for $I = \{1,2,\ldots\}$ and $r_1 \subseteq r_2 \subseteq \ldots$
Proof is by induction on the structure of the tree for expression $E(r)$ using monotonicity property. Case $E(r) = E_1(r) \circ E_2(r)$, assuming by inductive hypothesis $E_1(\bigcup_{i\in I} r_i) = \bigcup_{i\in I} E_1(r_i)$ and $E_2(\bigcup_{i\in I} r_i) = \bigcup_{i\in I} E_2(r_i)$.
Let $r = \bigcup_{i\in I} r_i$. We know by previous monotonicity argument that $\bigcup_{i\in I} E(r_i) \subseteq E(\bigcup_{i\in I} r_i)$, so we just need to show the converse direction. Let $(x,x') \in E(r)$ be arbitrary. We need to show $(x,x') \in \bigcup_{i\in I} E(r_i)$. Since $(x,x') \in E_1(r) \circ E_2(r)$, there exists z such that $(x,z) \in E_1(r)$ and $(z,x') \in E_2(r)$. By inductive hypothesis, $(x,z) \in \bigcup_{i\in I} E_1(r_i)$ and $(z,x') \in \bigcup_{i\in I} E_2(r_i)$. By definition of union there exists i_1, i_2 such that $(x,z) \in E_1(r_{i_1})$ and $(z,x') \in E_2(r_{i_2})$. Let $j = max(i_1, i_2)$. Then $r_{i_1} \subseteq r_j$ and $r_{i_2} \subseteq r_j$, so, by monotonicity of E_1 and E_2, $(x,z) \in E_1(r_j)$ and $(z,x') \in E_2(r_j)$. Thus, $(x,x') \in E_1(r_j) \circ E_2(r_j) = E(r_j)$ so $(x,x') \in \bigcup_{i\in I} E(r_i)$ as desired.

More on Mapping Code to Formulas

Local Mutable Variables

Assume our global variables are $V = \{x, z\}$
Program P introduces a local variable y inside a nested block:

$$x = x + 1; \{\textbf{var } y; y = x + 3; z = x + y + z\}; x = x + z$$

$R(P)$ should be a relation between (x, z) and (x', z').
Each statement should be relation between variables in scope. Inside the block we have variables $V_1 = \{x, y, z\}$. For assignment statement c: $z = x + y + z$,
$R(c)$ is a relation between x, y, z and x', y', z'.
Convention: consider the initial values of variables to be arbitrary
$R(y = x + 3; z = x + y + z) =$

Local Mutable Variables

Assume our global variables are $V = \{x, z\}$
Program P introduces a local variable y inside a nested block:

$$x = x + 1; \{\textbf{var } y; y = x + 3; z = x + y + z\}; x = x + z$$

$R(P)$ should be a relation between (x, z) and (x', z').
Each statement should be relation between variables in scope. Inside the block we have variables $V_1 = \{x, y, z\}$. For assignment statement c: $z = x + y + z$,
$R(c)$ is a relation between x, y, z and x', y', z'.
Convention: consider the initial values of variables to be arbitrary
$R(y = x + 3; z = x + y + z) = y' = x + 3 \wedge z' = 2x + 3 + z \wedge x' = x$

Local Mutable Variables

Assume our global variables are $V = \{x, z\}$
Program P introduces a local variable y inside a nested block:

$$x = x + 1; \{\textbf{var } y; y = x + 3; z = x + y + z\}; x = x + z$$

$R(P)$ should be a relation between (x, z) and (x', z').
Each statement should be relation between variables in scope. Inside the block we have variables $V_1 = \{x, y, z\}$. For assignment statement c: $z = x + y + z$,
$R(c)$ is a relation between x, y, z and x', y', z'.
Convention: consider the initial values of variables to be arbitrary
$R(y = x + 3; z = x + y + z) = y' = x + 3 \wedge z' = 2x + 3 + z \wedge x' = x$

$R(\{\textbf{var } y; y = x + 3; z = x + y + z\}) =$

Local Mutable Variables

Assume our global variables are $V = \{x, z\}$
Program P introduces a local variable y inside a nested block:

$$x = x + 1; \{\textbf{var } y; y = x + 3; z = x + y + z\}; x = x + z$$

$R(P)$ should be a relation between (x, z) and (x', z').
Each statement should be relation between variables in scope. Inside the block we have variables $V_1 = \{x, y, z\}$. For assignment statement c: $z = x + y + z$,
$R(c)$ is a relation between x, y, z and x', y', z'.
Convention: consider the initial values of variables to be arbitrary
$R(y = x + 3; z = x + y + z) = y' = x + 3 \wedge z' = 2x + 3 + z \wedge x' = x$

$R(\{\textbf{var } y; y = x + 3; z = x + y + z\}) = z' = 2x + 3 + z \wedge x' = x$

Local Variable Translation

$R_V(P)$ is formula for P in the scope that has the set of variables V. For example,

$$R_V(x = t) \;=\; x' = t \wedge \bigwedge_{v \in V \setminus \{x\}} v' = v$$

Then define

$R_V(\{var\ y; P\}) \;=$

Local Variable Translation

$R_V(P)$ is formula for P in the scope that has the set of variables V. For example,

$$R_V(x = t) \;=\; x' = t \wedge \bigwedge_{v \in V \setminus \{x\}} v' = v$$

Then define

$R_V(\{var\ y; P\}) \;=\; \exists y, y'.R_{V \cup \{y\}}(P)$

Local Variable Translation

$R_V(P)$ is formula for P in the scope that has the set of variables V. For example,

$$R_V(x = t) \;=\; x' = t \wedge \bigwedge_{v \in V \setminus \{x\}} v' = v$$

Then define

$R_V(\{var\ y; P\}) \;=\; \exists y, y'.R_{V \cup \{y\}}(P)$

Example: $R_{\{x,y,z\}}(x = y+1) = (x' = y+1 \wedge y' = y \wedge z = z')$, so

$$R_{\{x,z\}}(\{var\ y; x = y+1\}) \;=\; \exists y, y'.\ x' = y+1 \wedge y' = y \wedge z = z'$$

In the last formula we can eliminate y' (the result is that $y' = y$ disappears) and then eliminate y from $x' = y+1$ i.e. $y = x'-1$ (over integers). Thus the formula is equivalent to $z = z'$.

Local Variable Translation

$R_V(P)$ is formula for P in the scope that has the set of variables V. For example,

$$R_V(x = t) \;=\; x' = t \wedge \bigwedge_{v \in V \setminus \{x\}} v' = v$$

Then define

$R_V(\{var\ y; P\}) \;=\; \exists y, y'. R_{V \cup \{y\}}(P)$

Example: $R_{\{x,y,z\}}(x = y + 1) = (x' = y + 1 \wedge y' = y \wedge z = z')$, so

$$R_{\{x,z\}}(\{var\ y; x = y + 1\}) \;=\; \exists y, y'.\ x' = y + 1 \wedge y' = y \wedge z = z'$$

In the last formula we can eliminate y' (the result is that $y' = y$ disappears) and then eliminate y from $x' = y + 1$ i.e. $y = x' - 1$ (over integers). Thus the formula is equivalent to $z = z'$.

Exercise: express havoc(x) using var.

Local Variable Translation

$R_V(P)$ is formula for P in the scope that has the set of variables V. For example,

$$R_V(x = t) \;=\; x' = t \wedge \bigwedge_{v \in V \setminus \{x\}} v' = v$$

Then define

$R_V(\{var\ y; P\}) \;=\; \exists y, y'. R_{V \cup \{y\}}(P)$

Example: $R_{\{x,y,z\}}(x = y + 1) = (x' = y + 1 \wedge y' = y \wedge z = z')$, so

$$R_{\{x,z\}}(\{var\ y; x = y + 1\}) \;=\; \exists y, y'.\ x' = y + 1 \wedge y' = y \wedge z = z'$$

In the last formula we can eliminate y' (the result is that $y' = y$ disappears) and then eliminate y from $x' = y + 1$ i.e. $y = x' - 1$ (over integers). Thus the formula is equivalent to $z = z'$.

Exercise: express havoc(x) using var.

$$R_V(havoc(x)) \iff R_V(\{var\ y;\ x = y\})$$

Expressing Specifications as Commands

Shorthand: Havoc Multiple Variables at Once

Variables $V = \{x_1, \ldots, x_n\}$
Translation of $R(havoc(y_1, \ldots, y_m))$:

Shorthand: Havoc Multiple Variables at Once

Variables $V = \{x_1, \ldots, x_n\}$
Translation of $R(havoc(y_1, \ldots, y_m))$:

$$\bigwedge_{v \in V \setminus \{y_1, \ldots, y_m\}} v' = v$$

Exercise: the resulting formula is the same as for:

$$havoc(y_1); \ldots; havoc(y_m)$$

Thus, the order of distinct havoc-s does not matter.

Programs and Specs are Relations

program:	$x = x + 2; y = x + 10$
relation:	$\{(x, y, z, x', y', z') \mid x' = x + 2 \wedge y' = x + 12 \wedge z' = z\}$
formula:	$x' = x + 2 \wedge y' = x + 12 \wedge z' = z$

Specification:

$$z' = z \wedge (x > 0 \rightarrow (x' > 0 \wedge y' > 0)$$

Adhering to specification is relation subset:

$$\begin{array}{rl} & \{(x, y, z, x', y', z') \mid x' = x + 2 \wedge y' = x + 12 \wedge z' = z\} \\ \subseteq & \{(x, y, z, x', y', z') \mid z' = z \wedge (x > 0 \rightarrow (x' > 0 \wedge y' > 0))\} \end{array}$$

Non-deterministic programs are a way of writing specifications

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

$$z' = z \wedge x' > 0 \wedge y' > 0$$

Corresponding program:

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

$$z' = z \wedge x' > 0 \wedge y' > 0$$

Corresponding program:

$$\mathit{havoc}(x, y); \mathit{assume}(x > 0 \wedge y > 0)$$

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

$$z' = z \wedge x' > 0 \wedge y' > 0$$

Corresponding program:

$$havoc(x, y); assume(x > 0 \wedge y > 0)$$

Formula for relation:

$$z' = z \wedge x' > x \wedge y' > y$$

Corresponding program?

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

$$z' = z \wedge x' > 0 \wedge y' > 0$$

Corresponding program:

$$havoc(x, y); assume(x > 0 \wedge y > 0)$$

Formula for relation:

$$z' = z \wedge x' > x \wedge y' > y$$

Corresponding program?

Use local variables to store initial values.

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

$$z' = z \wedge x' > 0 \wedge y' > 0$$

Corresponding program:

$$havoc(x, y); assume(x > 0 \wedge y > 0)$$

Formula for relation:

$$z' = z \wedge x' > x \wedge y' > y$$

Corresponding program?

Use local variables to store initial values.

```
{ var x0; var y0;
  x0 = x; y0 = y;
  havoc(x,y);
  assume(x > x0 && y > y0)
}
```

Writing Specs Using Havoc and Assume

Global variables $V = \{x_1, \ldots, x_n\}$
Specification

$$F(x_1, \ldots, x_n, x_1', \ldots, x_n')$$

Becomes

Writing Specs Using Havoc and Assume

Global variables $V = \{x_1, \ldots, x_n\}$
Specification

$$F(x_1, \ldots, x_n, x_1', \ldots, x_n')$$

Becomes

$$\begin{array}{l} \{\ \mathit{var}\ y_1, \ldots, y_n; \\ \quad y_1 = x_1; \ldots; y_n = x_n; \\ \quad \mathit{havoc}(x_1, \ldots, x_n); \\ \quad \mathit{assume}(F(y_1, \ldots, y_n, x_1, \ldots, x_n))\ \} \end{array}$$

Program Refinement and Equivalence

For two programs, define **refinement** $P_1 \sqsubseteq P_2$ iff

$$R(P_1) \rightarrow R(P_2)$$

is a valid formula.
(Some books use the opposite meaning of $\sqsubseteq$.)
As usual, $P_2 \sqsupseteq P_1$ iff $P_1 \sqsubseteq P_2$.

- $P_1 \sqsubseteq P_2$ iff $\rho(P_1) \subseteq \rho(P_2)$

Define **equivalence** $P_1 \equiv P_2$ iff $P_1 \sqsubseteq P_2 \wedge P_2 \sqsubseteq P_1$

- $P_1 \equiv P_2$ iff $\rho(P_1) = \rho(P_2)$

Example for $V = \{x, y\}$

$$\{var\ x0; x0 = x; havoc(x); assume(x > x0)\} \sqsupseteq (x = x + 1)$$

Proof: Use R to compute formulas for both sides and simplify.

Program Refinement and Equivalence

For two programs, define **refinement** $P_1 \sqsubseteq P_2$ iff

$$R(P_1) \rightarrow R(P_2)$$

is a valid formula.
(Some books use the opposite meaning of $\sqsubseteq$.)
As usual, $P_2 \sqsupseteq P_1$ iff $P_1 \sqsubseteq P_2$.

- $P_1 \sqsubseteq P_2$ iff $\rho(P_1) \subseteq \rho(P_2)$

Define **equivalence** $P_1 \equiv P_2$ iff $P_1 \sqsubseteq P_2 \wedge P_2 \sqsubseteq P_1$

- $P_1 \equiv P_2$ iff $\rho(P_1) = \rho(P_2)$

Example for $V = \{x, y\}$

$$\{var\ x0; x0 = x; havoc(x); assume(x > x0)\} \sqsupseteq (x = x + 1)$$

Proof: Use R to compute formulas for both sides and simplify.

$$x' = x + 1 \wedge y' = y \ \rightarrow \ x' > x \wedge y' = y$$

Stepwise Refinement Methodology

Start form a possibly non-deterministic specification P_0
Refine the program until it becomes deterministic and efficiently executable.

$$P_0 \sqsupseteq P_1 \sqsupseteq \ldots \sqsupseteq P_n$$

Example:

$$\begin{aligned} & havoc(x); assume(x > 0); havoc(y); assume(x < y) \\ \sqsupseteq \quad & havoc(x); assume(x > 0); y = x + 1 \\ \sqsupseteq \quad & x = 42; y = x + 1 \\ \sqsupseteq \quad & x = 42; y = 43 \end{aligned}$$

In the last step program equivalence holds as well

Monotonicity with Respect to Refinement

Theorem: if $P_1 \sqsubseteq P_2$ then $(P_1; P) \sqsubseteq (P_2; P)$

Monotonicity with Respect to Refinement

Theorem: if $P_1 \sqsubseteq P_2$ then $(P_1; P) \sqsubseteq (P_2; P)$
Version for relations: $(p_1 \subseteq p_2) \rightarrow (p_1 \circ p) \subseteq (p_2 \circ p)$

Monotonicity with Respect to Refinement

Theorem: if $P_1 \sqsubseteq P_2$ then $(P_1; P) \sqsubseteq (P_2; P)$
Version for relations: $(p_1 \subseteq p_2) \rightarrow (p_1 \circ p) \subseteq (p_2 \circ p)$

Theorem: if $P_1 \sqsubseteq P_2$ then $(P; P_1) \sqsubseteq (P; P_2)$

Monotonicity with Respect to Refinement

Theorem: if $P_1 \sqsubseteq P_2$ then $(P_1; P) \sqsubseteq (P_2; P)$
Version for relations: $(p_1 \subseteq p_2) \rightarrow (p_1 \circ p) \subseteq (p_2 \circ p)$

Theorem: if $P_1 \sqsubseteq P_2$ then $(P; P_1) \sqsubseteq (P; P_2)$
Version for relations: $(p_1 \subseteq p_2) \rightarrow (p \circ p_1) \subseteq (p \circ p_2)$

Theorem: if $P_1 \sqsubseteq P_2$ and $Q_1 \sqsubseteq Q_2$ then

$$(\mathit{if}\ (*) P_1\ \mathit{else}\ Q_1) \sqsubseteq (\mathit{if}\ (*) P_2\ \mathit{else}\ Q_2)$$

Monotonicity with Respect to Refinement

Theorem: if $P_1 \sqsubseteq P_2$ then $(P_1; P) \sqsubseteq (P_2; P)$
Version for relations: $(p_1 \subseteq p_2) \rightarrow (p_1 \circ p) \subseteq (p_2 \circ p)$

Theorem: if $P_1 \sqsubseteq P_2$ then $(P; P_1) \sqsubseteq (P; P_2)$
Version for relations: $(p_1 \subseteq p_2) \rightarrow (p \circ p_1) \subseteq (p \circ p_2)$

Theorem: if $P_1 \sqsubseteq P_2$ and $Q_1 \sqsubseteq Q_2$ then

$$(\mathit{if}\ (*) P_1\ \mathit{else}\ Q_1) \sqsubseteq (\mathit{if}\ (*) P_2\ \mathit{else}\ Q_2)$$

Version for relations: $(p_1 \subseteq p_2) \wedge (q_1 \subseteq q_2) \ \rightarrow \ (p_1 \cup q_1) \subseteq (p_2 \cup q_2)$

Loops

Loops: Example

Consider the set of variables $V = \{x, y\}$ and this program L:

while $(\mathrm{x} > 0)$ {
  $\mathrm{x} = \mathrm{x} - \mathrm{y}$
}

When the loop terminates, what is the (smallest) relation $\rho(L)$ between state (x, y) before loop started executing and the final state (x', y')?

Loops: Example

Consider the set of variables $V = \{x, y\}$ and this program L:

while (x $>$ 0) {
  x = x − y
}

When the loop terminates, what is the (smallest) relation $\rho(L)$ between state (x, y) before loop started executing and the final state (x', y')?
Let k be the number of times loop executes.

Loops: Example

Consider the set of variables $V = \{x, y\}$ and this program L:

while (x $>$ 0) {
  x = x − y
}

When the loop terminates, what is the (smallest) relation $\rho(L)$ between state (x, y) before loop started executing and the final state (x', y')?
Let k be the number of times loop executes.

- $k = 0$:

Loops: Example

Consider the set of variables $V = \{x, y\}$ and this program L:

while (x $>$ 0) {
  x = x − y
}

When the loop terminates, what is the (smallest) relation $\rho(L)$ between state (x, y) before loop started executing and the final state (x', y')?
Let k be the number of times loop executes.

- $k = 0$: $x \leq 0 \wedge x' = x \wedge y' = y$

Loops: Example

Consider the set of variables $V = \{x, y\}$ and this program L:

while (x $>$ 0) {
  x = x − y
}

When the loop terminates, what is the (smallest) relation $\rho(L)$ between state (x, y) before loop started executing and the final state (x', y')?
Let k be the number of times loop executes.

- $k = 0$: $x \leq 0 \wedge x' = x \wedge y' = y$
- $k = 1$:

Loops: Example

Consider the set of variables $V = \{x, y\}$ and this program L:

```
while (x > 0) {
  x = x − y
}
```

When the loop terminates, what is the (smallest) relation $\rho(L)$ between state (x, y) before loop started executing and the final state (x', y')?
Let k be the number of times loop executes.

- $k = 0$: $x \leq 0 \wedge x' = x \wedge y' = y$
- $k = 1$: $x > 0 \wedge x' = x - y \wedge y' = y \wedge x' \leq 0$

Loops: Example

Consider the set of variables $V = \{x, y\}$ and this program L:

while (x $>$ 0) {
  x = x − y
}

When the loop terminates, what is the (smallest) relation $\rho(L)$ between state (x, y) before loop started executing and the final state (x', y')?
Let k be the number of times loop executes.

- $k = 0$: $x \leq 0 \wedge x' = x \wedge y' = y$
- $k = 1$: $x > 0 \wedge x' = x - y \wedge y' = y \wedge x' \leq 0$
- $k > 0$:

Loops: Example

Consider the set of variables $V = \{x, y\}$ and this program L:

while $(\mathrm{x} > 0)$ {
  $\mathrm{x} = \mathrm{x} - \mathrm{y}$
}

When the loop terminates, what is the (smallest) relation $\rho(L)$ between state (x, y) before loop started executing and the final state (x', y')?
Let k be the number of times loop executes.

- $k = 0$: $x \leq 0 \wedge x' = x \wedge y' = y$
- $k = 1$: $x > 0 \wedge x' = x - y \wedge y' = y \wedge x' \leq 0$
- $k > 0$: $x > 0 \wedge x' = x - ky \wedge x' \leq 0 \wedge y' = y$

Solution:

$$(x \leq 0 \wedge x' = x \wedge y' = y) \vee$$
$$(\exists k.\ k > 0 \wedge x > 0 \wedge x' = x - ky \wedge x' \leq 0 \wedge y' = y)$$

Heuristically Eliminating a Quantifier from formula

$$\exists k.\ k > 0 \wedge x > 0 \wedge x' = x - ky \wedge x' \leq 0 \wedge y' = y$$

Heuristically Eliminating a Quantifier from formula

$$\exists k.\ k > 0 \wedge x > 0 \wedge x' = x - ky \wedge x' \leq 0 \wedge y' = y$$

$$\exists k.\ k > 0 \wedge x > 0 \wedge ky = x - x' \wedge x' \leq 0 \wedge y' = y$$

Heuristically Eliminating a Quantifier from formula

$$\exists k.\ k > 0 \wedge x > 0 \wedge x' = x - ky \wedge x' \leq 0 \wedge y' = y$$

$$\exists k.\ k > 0 \wedge x > 0 \wedge ky = x - x' \wedge x' \leq 0 \wedge y' = y$$

Note that $x - x' > 0$ and $k > 0$ so from $ky = x - x'$ we get $y > 0$.

Heuristically Eliminating a Quantifier from formula

$$\exists k.\ k>0 \wedge x>0 \wedge x' = x - ky \wedge x' \leq 0 \wedge y' = y$$

$$\exists k.\ k>0 \wedge x>0 \wedge ky = x - x' \wedge x' \leq 0 \wedge y' = y$$

Note that $x - x' > 0$ and $k > 0$ so from $ky = x - x'$ we get $y > 0$.

$$\exists k.\ k>0 \wedge y>0 \wedge x>0 \wedge y | (x - x') \wedge k = (x - x')/y \wedge x' \leq 0 \wedge y' = y$$

Apply one-point rule to eliminate k

Heuristically Eliminating a Quantifier from formula

$$\exists k.\ k>0 \wedge x>0 \wedge x' = x - ky \wedge x' \leq 0 \wedge y' = y$$

$$\exists k.\ k>0 \wedge x>0 \wedge ky = x - x' \wedge x' \leq 0 \wedge y' = y$$

Note that $x - x' > 0$ and $k > 0$ so from $ky = x - x'$ we get $y > 0$.

$$\exists k.\ k>0 \wedge y>0 \wedge x>0 \wedge y | (x-x') \wedge k = (x-x')/y \wedge x' \leq 0 \wedge y' = y$$

Apply one-point rule to eliminate k

$$((x-x')/y) > 0 \wedge y>0 \wedge x>0 \wedge y | (x-x') \wedge x' \leq 0 \wedge y' = y$$

which is also equivalent to simply

Heuristically Eliminating a Quantifier from formula

$$\exists k.\ k>0 \wedge x>0 \wedge x' = x - ky \wedge x' \leq 0 \wedge y' = y$$

$$\exists k.\ k>0 \wedge x>0 \wedge ky = x - x' \wedge x' \leq 0 \wedge y' = y$$

Note that $x - x' > 0$ and $k > 0$ so from $ky = x - x'$ we get $y > 0$.

$$\exists k.\ k>0 \wedge y>0 \wedge x>0 \wedge y | (x-x') \wedge k = (x-x')/y \wedge x' \leq 0 \wedge y' = y$$

Apply one-point rule to eliminate k

$$((x-x')/y) > 0 \wedge y>0 \wedge x>0 \wedge y | (x-x') \wedge x' \leq 0 \wedge y' = y$$

which is also equivalent to simply

$$y>0 \wedge x>0 \wedge y | (x-x') \wedge x' \leq 0 \wedge y' = y$$

Formula for Loop

Meaning of

while $(x > 0)$ {
  $x = x - y$
}

is given by formula

$$(x \leq 0 \wedge x' = x \wedge y' = y) \vee$$
$$(y > 0 \wedge x > 0 \wedge y | (x - x') \wedge x' \leq 0 \wedge y' = y)$$

Formula for Loop

Meaning of

while $(x > 0)$ {
  $x = x - y$
}

is given by formula

$$(x \leq 0 \wedge x' = x \wedge y' = y) \vee$$
$$(y > 0 \wedge x > 0 \wedge y | (x - x') \wedge x' \leq 0 \wedge y' = y)$$

What happens if initially $x > 0 \wedge y \leq 0$?

- in the formula
- in the program

Integer Programs with Loops

Integer programs with loops are Turing complete and can compute all computable functions (we can use large integers as Turing machine tape).

Even if we cannot find a closed-form integer arithmetic formula, we may be able to find

- a formula in a richer logic
- a property of the meaning of the loop
  (e.g. formula for the superset)

To help with these tasks, we give mathematical semantics of loops
Useful concept for this is transitive closure: $r^* = \bigcup_{n \geq 0} r^n$
(We may or may not have a general formula for r^n or r^*)

Some facts about relations

Let $r \subseteq S \times S$ and $\Delta = \{(x,x) \mid x \in S\}$. Then

$$\Delta \circ r = r = r \circ \Delta$$

We say that r is **reflexive** iff $\forall x \in S.(x,x) \in r$.

- equivalently, reflexivity means $\Delta \subseteq r$

Relation r is **transitive** iff

$$\forall x, y, z.\ ((x,y) \in r \wedge (y,z) \in r \rightarrow (x,z) \in r)$$

which is the same as saying $r \circ r \subseteq r$

Transitive Closure of a Relation

$r \subseteq S \times S$. Define $r^0 = \Delta$ and $r^{n+1} = r \circ r^n$. Then $(x_0, x_n) \in r^n$ iff $\exists x_1, \ldots, x_{n-1}$ such that $(x_i, x_{i+1}) \in r$ for $0 \leq i \leq n-1$.
Define reflexive transitive closure of r by

$$r^* = \bigcup_{n \geq 0} r^n$$

Properties that follow from the definition:

- $(x_0, x_n) \in r^*$ iff there exists $n \geq 0$ and $\exists x_1, \ldots, x_{n-1}$ such that $(x_i, x_{i+1}) \in r$ for $0 \leq i \leq n-1$ (a path in the graph r)
- r^* is a reflexive and transitive relation
- If s is a reflexive transitive relation and $r \subseteq s$, then $r^* \subseteq s$
  - r^* is the smallest reflexive transitive relation containing r
- $(r^{-1})^* = (r^*)^{-1}$
- $r_1 \subseteq r_2$ implies $r_1^* \subseteq r_2^*$
- $r^* = \Delta \cup (r \circ r^*)$ and, likewise, $r^* = \Delta \cup (r^* \circ r)$

Towards meaning of loops: unfolding

Loops can describe an infinite number of basic paths
(for a larger input, program takes a longer path)
Consider loop

$$L \equiv \textbf{while}(F)c$$

We would like to have

$$\begin{aligned} L &\equiv \textbf{if}(F)\ (c; L) \\ &\equiv \textbf{if}(F)\ (c; \textbf{if}(F)\ (c; L)) \end{aligned}$$

For $r_L = \rho(L)$, $r_c = \rho(c)$, $\Delta_1 = \Delta_{\tilde{F}}$, $\Delta_2 = \Delta_{\tilde{\neg F}}$ we have

$$\begin{aligned} r_L &= (\Delta_1 \circ r_c \circ r_L) \cup \Delta_2 \\ &= (\Delta_1 \circ r_c \circ ((\Delta_1 \circ r_c \circ r_L) \cup \Delta_2)) \cup \Delta_2 \\ &= \Delta_2 \cup \\ &\quad (\Delta_1 \circ r_c) \circ \Delta_2 \cup \\ &\quad (\Delta_1 \circ r_c)^2 \circ r_L \end{aligned}$$

Unfolding Loops

$$\begin{aligned} r_L \;=\; & \Delta_2 \cup \\ & (\Delta_1 \circ r_c) \circ \Delta_2 \cup \\ & (\Delta_1 \circ r_c)^2 \circ \Delta_2 \cup \\ & (\Delta_1 \circ r_c)^3 \circ r_L \end{aligned}$$

We prove by induction that for every $n \geq 0$,

$$(\Delta_1 \circ r_c)^n \circ \Delta_2 \subseteq r_L$$

So, $\bigcup_{n\geq 0}\big((\Delta_1 \circ r_c)^n \circ \Delta_2\big) \subseteq r_L$, that is

$$\Big(\bigcup_{n\geq 0}(\Delta_1 \circ r_c)^n\Big) \circ \Delta_2 \subseteq r_L$$

We do not wish to have unnecessary elements in relation, so we try

$$r_L = (\Delta_1 \circ r_c)^* \circ \Delta_2$$

and this does satisfy $r_L = (\Delta_1 \circ r_c \circ r_L) \cup \Delta_2$, so we define

$$\rho(\textbf{while}(F)c) = (\Delta_{\tilde{F}} \circ \rho(c))^* \circ \Delta_{\neg\tilde{F}}$$

Why loop semantics satisfies the condition

We defined

$$r_L = (\Delta_1 \circ r_c)^* \circ \Delta_2$$

Show that $(\Delta_1 \circ r_c \circ r_L) \cup \Delta_2$ equals r_L, as we expect from recursive definition of a while loop.

Why loop semantics satisfies the condition

We defined

$$r_L = (\Delta_1 \circ r_c)^* \circ \Delta_2$$

Show that $(\Delta_1 \circ r_c \circ r_L) \cup \Delta_2$ equals r_L, as we expect from recursive definition of a while loop.

Using property $r^* = \Delta \cup r \circ r^*$ we have

$$\begin{aligned} r_L &= (\Delta_1 \circ r_c)^* \circ \Delta_2 \\ &= [\Delta \cup \Delta_1 \circ r_c \circ (\Delta_1 \circ r_c)^*] \circ \Delta_2 \\ &= \Delta_2 \cup [\Delta_1 \circ r_c \circ (\Delta_1 \circ r_c)^* \circ \Delta_2] \\ &= \Delta_2 \cup \Delta_1 \circ r_c \circ r_L \end{aligned}$$

Using Loop Semantics in Example

ρ of L:

while $(x > 0)$ {
  $x = x - y$
}

is:

Using Loop Semantics in Example

ρ of L:

while (x $>$ 0) {
  x = x − y
}

is:

$$(\Delta_{\tilde{x>0}} \circ \rho(x = x - y))^* \circ \Delta_{\neg(\tilde{x>0})}$$

Compute each relation:

$$\begin{aligned}
\Delta_{\tilde{x>0}} &= \{((x,y),(x,y)) \mid x > 0\} \\
\Delta_{\neg(\tilde{x>0})} &= \{((x,y),(x,y)) \mid x \leq 0\} \\
\rho(x = x - y) &= \{((x,y),(x-y,y)) \mid x,y \in \mathbb{Z}\} \\
\Delta_{\tilde{x>0}} \circ \rho(x = x - y) &= \\
(\Delta_{\tilde{x>0}} \circ \rho(x = x - y))^k &= \\
(\Delta_{\tilde{x>0}} \circ \rho(x = x - y))^* &= \\
\rho(L) &=
\end{aligned}$$